ViewControllerImpl

904

Methods

Name	Declaration	Description
addViewListener	public final void addViewListener (ViewListener listener)	Add a ViewListener.
clear	public void clear()	Clear local state by setting the data reference to null and by removing all ViewListeners.
exit	public void exit()	Get read to exit. Clear local state by setting the data reference to null, removing all Viewlisteners and setting view listeners to null.
fireViewEvent	public final void fireViewEvent(ViewEvent event)	If the ViewEvent is not null then send it to all ViewListeners
getComponent	public Component getComponent()	Return the Component that is "this" ViewController. By default, "this" is returned. Redefine this method in ViewControllerBaseImpl when you have a non-java.awt.Component superclass.
getJTCs	public Vector getJTCs()	Return all JTC type objects defined. By default null is returned. Typically, ViewControllers will not return anything.
getPermissions	public String[] getPermissions()	Return a set "keys" that can a management system can use when assigning JTC function based on roles (i.e. group, user). For example, consider the common case of operator override. In grocery store, if a cashier makes a mistake, a manager inserts a key or enters a password to enable more function on the cash register. The software analogy is that a button may become active or disabled. Suppose the ViewController implements a button labeled "Override" and it is the only component whose state can be visibly altered outside the ViewController. The ViewController writer will return: "Override" In this case, the only options are ENABLE or DISABLE. Suppose these constants are define to be 0x001 and 0x002, respectively. A management system that maintains user privileges is queried at runtime. The ViewController is then called with setPermissions(keys, values) where keys is "Override" and values is "0x001". The ViewController writer now responds to this request by turning off the button. Instead of hard coding the possible roles, the ViewController simply reacts to key/value settings. By default, nothing is returned.

FIG. 9C

ViewControllerImpl (continued)

904

Methods

Name	Declaration	Description
init	public void init()	Initialize, by default do nothing.
isEnabled	public boolean isEnabled()	Is this ViewController enabled?
isValid	public boolean isValid()	Is the ViewController in a consistent state? This usually means: Do all fields pass ValidationRules? The meaning could also be application specific. This value can provide other components with the ability to show a visual indicator, such as an X or a check in a tree menu indicating incomplete or partial data. The default value is true.
isVisible	public boolean isVisible()	Is this ViewController visible?
refresh	public void refresh(Object data)	Data objects are being passed in. By default, keep a reference to them. Interpretation of the data is performed in the subclass. For example, suppose the data being passed is a Customer object. Then a subclass can perform the following: This can be extended to more complex data types and data type composites (i.e. arrays, Vectors, etc.).
removeViewListener	public final void removeViewListener (ViewListener listener)	Remove a ViewListener.
setEnabled	public void setEnabled(boolean toggle)	Enable or disable the ViewController. Remember the state and ask the ViewControllerBaseImpl to handle it.

FIG. 9D

ViewControllerImpl (continued)

FIG. 9E

904

Methods

Name	Declaration	Description

setPermissions	public void setPermissions (Hashtable permissions)	Given a set of keys and values, update the internal state of the ViewController. The keys and values are supplied via a management system and relate to roles (i.e. users and groups). The possible values in the key/value pairs are application and ViewController specific. For example, create an interface to define the keys and values:
		public interface Customer {
		public static final String DETAILS="DETAILS";
		public static final String ON="1";
		public static final String OFF="0";
		}
		then set the ViewController:
		Hashtable permissions=new Hashtable();
		permissions.put(Customer.DETAILS, Customer.ON);
		vc.setPermissions(Permissions);
		The ViewController will interpret the meaning of ON and perform the necessary action, such as active a button. The meaning of keys, values and actions should be defined in a GUI spec. By default, nothing happens.

setProperties	public void setProperties(Properties properties)	Set the properties. Default is to do nothing.
setResources	public void setResources(ResourceBundle bundle)	Set the ResourceBundles. Default is to do nothing.
setValidationLevel	public void setValidationLevel(int level)	Set the validation level to indicate when ValidationRules should be applied Four constants are defined in the ValidationRule class: NONE COMPONENT FOCUS VIEWEVENT This value will be stored for the subclass to reference and act. The default value is ValidationRule.NONE.
setVisible	public void setVisible(boolean visible)	Set the ViewController's visibility on or off. Remember the state and ask the ViewControllerBaseImpl to handle it.
toString	public String toString()	Return the instance class name.

ApplicationMediatorImpl (continued) *FIG. 17D* 1704

Methods

Name	Declaration	Description
getPermissions	public String[] getPermissions()	Get the settable permission keys. By default, return the class names of all allocated ViewControllers and ApplicationMediators.
getVC	protected ViewController getVC(int i)	Return the i'th ViewController
init	public void init()	Initialize the ApplicationMediator, nothing to do by default.
init	public void init(ApplicationMediator applicationMediator)	Initialize the ApplicationMediator using the listeners of an existing ApplicationMediator.
initApplicationMediators	public final void initApplicationMediators(String classnames[]) throws ClassNotFoundException, InstantiationException, IllegalAccessException	For each ApplicationMediator classname, load it, new it and add myself as a ViewEvent. The Factory class is used as helper class.
initViewControllers	public final void initViewControllers(String classnames[]) throws ClassNotFoundException, InstantiationException, IllegalAccessException	For each ViewController classname, load it, new it and add myself as a ViewEvent. The Factory class is used as helper class.
isEnabled	public boolean isEnabled()	Is the ApplicationController enabled?
isValid	public boolean isValid()	Return the AND'ed value of calling isValid on ApplicationMediators and ViewControllers.
isVisible	public boolean isVisible()	Is the ApplicationController visible? Hardly, since it is a non visible class. But this looks to see if any of its ViewControllers are visible. Not really, they were all set to visible/invisible via the setVisible method and we remembered the state to return here.
processViewEvent	public abstract void processViewEvent(ViewEvent e)	Deliver the ViewEvent to the subclass via this method.
refresh	public void refresh(Oject data)	When new data arrives allow the ViewControllers and ApplicationControllers to be refreshed also.
removePlacementListener	public final void removePlacementListener(PlacementListener listener)	Removes the PlacementListener.

AUS920033945US1

FIG. 17E

1704

Methods

Name	Declaration	Description
removeRequestListener	public final void removeRequestListener(RequestListener listener)	Removes the RequestListener.
removeViewListener	public final void removeViewListener(ViewListener listener)	Removes the ViewListener.
requestException	public void requestException(RequestException yikes)	Called back because an asynchronous request has thrown an Exception. By default, print the message to System.err.
requestResponse	public void requestResponse(RequestEvent response)	Called back with the results of an asynchronous request. By default, call refresh with the data in the response.
run2	public final void run2()	This method is used in style 1 threading. Rename this to run() and uncomment the code as described in the class javadoc.
setAM	public void setAM(ApplicationMediator applicationMediator, int i)	Set the i'th ApplicationMediator.
setEnabled	public void setEnabled(boolean toggle)	Call setEnabled on each ViewController and ApplicationMediator.
setPermissions	public void setPermissions(Hashtable permissions)	Set the permissions. By default, call setPermissions on each ViewController and ApplicationMediator.
setProperties	public void setProperties(Properties properties)	Set the properties. By default, call setProperties on each ViewController and ApplicationMediator.
setResources	public void setResources(ResourceBundle bundle)	Set the resources. By default, call setResources on each ViewController and ApplicationMediator.
setVC	public void setVC(ViewController viewController, int i)	Set the i'th ViewController.
setVisible	public void setVisible(boolean visible)	Set visible on each ViewController and ApplicationMediator.
toString	public String toString()	Return the Class name of the ApplicationController instance.
viewEventPerformed	public void viewEventPerformed(ViewEvent e)	A ViewEvent is delivered. Process it using Threading style 1 or 2. In the end, the processViewEvent will be called on the subclass.

AUS 9990339US1

1706

ApplicationMediatorImpl.exit(): AUS8-1999-0694

```
/**
 * Exit the ApplicationMediator by exiting all allocated ViewControllers
 * and ApplicationMediators. All data is set to null, and lists are
 * destroyed. An 'exited' ApplicationMediator cannot be used again.
 * If this method is overriden in a subclass, be sure to invoke
 * super.exit();
 **/
public void exit() {
    synchronized (this) {

        /* Used for style 1 event dispatching. Leave this code commented. */
        //if (this.eventThread !=null) {
        //    try {
        //        eventThread.stop ();
        //    } catch (Exception e) {
        //    }
        //}

        /* Used for style 2 event dispatching. Leave this code commented. */
        for (int i = 0; i < runningThreads.size(); i++) {
            ((ApplicationMediatorThread) runningThreads.elementAt (i)) .stop();
        }
        runningThreads.removeAllElements();
        viewListeners.removeAllElements();
        try {
            for (int i = 0; i < viewControllers.size(); i++) {
                ((ViewController) viewControllers.elementAt(i)) .setEnabled(false);
                ((ViewController) viewControllers.elementAt(i)) .exit ();
            }
            for (int i = 0; i < applicationMediators.size(); i++) {
                ((ApplicationMediator) applicationMediators.elementAt(i)) .setEnabled(false);
                ((ApplicationMediator) applicationMediators.elementAt(i)) .exit();
            }
        } catch (Exception noProblem) {
        }
        viewControllers = null;
        applicationMediators = null;
        runningThreads = null;
        runningThreads = null;
        data = null;
    }
}
```

FIG. 17F

AUS 9990339US1

1708

ApplicationMediatorImpl.clear(): AUS8-1999-0694

```
/**
 * Clear the ApplicationMediator by clearing all allocated ViewControllers
 * and ApplicationMediators. All data is set to null, but lists are
 * not destroyed. A 'cleared' ApplicationMediator can be used again.
 * If this methed is overriden in a subclass, be sure to invoke
 * super.clear();
 */
public void clear() {
    synchronized (this) {

        /* Used for style 1 event dispatching. Leave this code commented. */
        //if (this.eventThread != null) {
        //    try {
        //        eventThread.stop ();
        //    } catch (Exception e) {
        //    }
        //}

        /* Used for style 2 event dispatching. Leave this code commented. */
        for (int i = 0; i < runningThreads.size(); i++) {
            ((ApplicationMediatorThread) runningThreads.elementAt (i)) .stop();
        }
        runningThreads.removeAllElements();

        //
        try {
            for (int i = 0; i < viewControllers.size(); i++) {
                ((ViewController) viewControllers.elementAt(i)) .setEnabled(false);
                ((ViewController) viewControllers.elementAt(i)) .clear ();
            }
            for (int i = 0; i < applicationMediators.size(); i++) {
                ((ApplicationMediator) applicationMediators.elementAt(i)) .setEnabled(false);
                ((ApplicationMediator) applicationMediators.elementAt(i)) .clear();
            }
        } catch (Exception noRealProblem) {
        }
        viewControllers = null;
        applicationMediators = null;
        data = null;
        viewListeners.removeAllElements();
    }
}
```

FIG. 17G

1710

```
/**
 * Initalize the ApplicationMediator using the listeners of an
 * existing ApplicationMediator.
 */
public void init(ApplicationMediator applicationMediator) {
        if (applicationMediator instanceof ApplicationMediatorImpl) {
                ApplicationMediatorImpl a = (ApplicationMediatorImpl) applicationMediator;
                requestListeners = (Vector) a.requestListeners.clone();
                placementListeners = (Vector) a.placementListeners.clone();
                topListeners = (Vector) a.topListeners.clone();
                addViewListener(a);
        }
        init();
}
```

FIG. 17H

1712

```
/**
 * When new data arrives allow the ViewControllers
 * and ApplicationControllers to be refreshed also.
 */
public void refresh(Object data) {
        this.data = data;
        try {
                synchronized (viewControllers) {
                        for (int j = 0; j < viewControllers.size(); j++) {
                                ((ViewController) viewControllers.elementAt(j)).
                                        refresh(data);
                        }
                }
        } catch (Exception noRealProblem) {
        }
        try {
                synchronized (applicationMediators) {
                        for (int j = 0; j < applicationMediators.size(); j++) {
                                ((ApplicationMediator) applicationMediators.
                                        elementAt(j)).refresh(data);
                        }
                }
        } catch (Exception noRealProblem) {
        }
}
```

FIG. 17I

AUS 990339 US1

1714

```
/**
 * A ViewEvent is delivered. Process it using Threading style 1 or 2. In
 * the end, the processViewEvent will be called on the subclass.
 */
public void viewEventPerformed (ViewEvent e) {
      /* Used for style 2 event dispatching, start on inner class thread */
      ApplicationMediatorThread t = new ApplicationMediatorThread (e);
      runningThreads.addElement (t);
      t.start ();

      /* Used for style 1 event dispatching. Leave this code commented. */
      //ViewEvent saved = saveViewEvent(e);
      //if (eventThread == null || !eventThread.isAlive()) {
      // finished = false;
      // eventThread = new Thread(this);
      // eventThread.start ();
      //}
      //synchronized (this) {
      // notify();
      //}
}
```

FIG. 17J

1714

```
/**
 * This method is used in style 1 threading. Rename this to run ()
 * and uncomment the code as described in the class javadoc.
 */
public final void run2 () {
    /* Used for style 1 event dispatching. Leave this code commented. */
    /*
    while (true) {
            ViewEvent event = null;
        event = getViewEvent ();
        if (event != null) {
            handleViewEvent (event);
        } else {
            waitForEvent ();
            if (finished) {
            // something went wrong with the thread so hose this loop
            break;
            }
        }
    }
    */
}
```

FIG. 17K

1714

```
/**
* Private class to handle executions of ViewEvents () on another thread.
**/
private class ApplicationMediatorThread extends Thread {
    /**
    * The current event
    **/
    private ViewEvent event;
    /**
    * Create an ApplicationMediatorThread to process the ViewEvent
    **/
    public ApplicationMediatorThread(ViewEvent event) {
        super ();
        this.event = event;
    }
    /**
    * Just call the handleViewEvent method that the subclass will override
    **/
    public void run () {
        processViewEvent (event);
    }
}
```

FIG. 17L

1714

```
/**
 * Save the current ViewEvent on a Q
 */
private final ViewEvent saveViewEvent (ViewEvent e) {
    /* Used for style 1 event dispatching. Leave this code commented. */
    //return viewEventQueue.add(e);
    return null;
}

/**
 * Method: return the first view event saved. Used by the Q'ing system.
 */
private ViewEvent getViewEvent () {
    /* Used for style 1 event dispatching. Leave this code commented. */
    //return (ViewEvent) viewEventQueue.remove();
    return null;
}
```

FIG. 17M

Transporter

FIG. 26C

2604

Methods

Name	Declaration	Description
addDestinationListener	public void addDestinationListener (Object major, Destination destination)	Add the Destination using the given major code. If the destination is present with the same major don't re-add it - only one major/destination pair can exist. If the major is present, but the destination isn't, add the destination to the list of other destinations with the same key. If the key isn't present, store it and then add the new destination. If the destination is disabled, do nothing.
clear	public void clear()	For each RequestEvent not started, a RequestException will be thrown and the internal data structures will be emptied including RequestEvent queues and listeners.
exit	public void exit()	For each RequestEvent not started, a RequestException will be thrown and the internal data structures will be emptied including RequestEvent queues and listeners. All variable references will be set to null.
getDestinations	public synchronized Vector getDestinations()	Return a Vector of all Destinations currently registered.
getDestinations	public Vector getDestinations(Object major)	Return a Vector of the Destinations currently registered for the given major code.
getJTCs	public Vector getJTCs()	Return allocated JTC objects. By default, return the Destinations.
getMajorCodes	public Vector getMajorCodes()	Return a Vector of the registered major codes.
init	public void init()	Initialize the transporter. By default, do nothing.
isEnabled	public boolean isEnabled()	Is this Transporter enabled or disabled? A Transporter that is disabled will not process a RequestEvents but will throw RequestExceptions.

AUS920033911

Transporter (continued)

FIG. 26D

2604

Methods

Name	Declaration	Description
isTagging	public boolean isTagging()	Is this Transporter tagging RequestEvents?
processDestinations	protected void processDestinations(RequestEvent request, Vector currentDestinations) throws RequestException	Given a RequestEvent and a Vector of destinations, call each Destination in FIFO/FEFR order. If tagging is enabled, then append a status tag to the RequestEvent.
removeDestinationListener	public void removeDestinationListener (Object major, Destination d)	Remove the destination using the given major. If the destination is not present, do nothing. If the destination is present, just remove it. If it was the last destination, remove all references to the major code.
requestEventPerformed	public void requestEventPerformed(RequestEvent request) throws RequestException	Submit a synchronous request. For each Destination that is listening for the current family of RequestEvents (the major code), send the RequestEvent to the Destination for processing. If there is a problem, throw a RequestException. Continue processing the RequestEvent as long as a RequestException is not thrown by a Destination and the RequestEvent is not consumed. If tagging is enabled, then append a status tag to the RequestEvent. Destinations are processed in the following FIFO order: 1–All using "!" (priority). 2–All using a major code. 3–All using "*".
requestEventPerformed	public void requestEventPerformed(RequestEvent request, RequestResponseListener caller) throws RequestException	Submit an asynchronous request. See the synchronous requestEventPerformed for more information.
setEnabled	public void setEnabled(boolean toggle)	Enable or disable the Transporter. A disabled Transporter will throw RequestExceptions if accessed via requestEventPerformed.
setRequestTagging	public void setRequestTagging(boolean toggle)	Stop or start the tagging of Requests.
toString	public String toString()	Return the String Transporter plus the number of registered Destinations.

2604

Transporter.processDestinations(RequestEvent, Vector):AUS8-1999-0693

```
/**
 * Given a RequestEvent and a Vector of destinations, call each Destination
 * in FIFO/FEFR order.
 * <p>
 * If tagging is enabled, then append a status tag to the RequestEvent.
 * @exception RequestException if the Request can't be submitted
 */
protected void processDestinations(RequestEvent request, Vector currentDestinations) throws RequestException {
      if (!enabled) {
            throw new RequestException("Transporter disabled");
      }
      if (currentDestinations == null)
            return;

      /* process FIFO/FEFR */
      Destination d = null;
      int size = currentDestinations.size();
      for (int i = 0; !request.isConsumed() && i < size; i++) {
            d = (Destination) currentDestinations.elementAt(i);
            d.requestEventPerformed(request);
            /* Try to tag the request */
            if (tagging)
                  request.setStatus (request.getStatus() + d);
      }
}
```

FIG. 26E

AUS 9 2 0339 US11

Transporter.requestEventPerformed(RequestEvent):AUS8-1999-0693

```
/**
 * Submit a synchronous request. For each Destination that is listening for
 * the current family of RequestEvents (the major code), send the RequestEvent
 * to the Destination for processing. If there is a problem, throw
 * a RequestException. Continue processing the RequestEvent as long
 * as a RequestException is not thrown by a Destination and the RequestEvent
 * is not consumed.
 * <p>
 * If tagging is enabled, then append a status tag to the RequestEvent.
 *<p>
 * Destinations are processed in the following FIFO order:
 * 1- All using "!" (priority).
 * 2- All using a major code.
 * 3- All using "*".
 *<p>
 * @exception RequestException if the Request can't be submitted
 */
public void requestEventPerformed(RequestEvent request) throws RequestException {
     if (!enabled) {
          throw new RequestException("Transporter disabled");
     }

     /* Try to tag the request */                                          2606
     if (tagging)
          request.setStatus(request.getStatus() + "[Transporter]");
     /* Process PRIORITY, major and then WILDCARD destinations */
     processDestinations(request, getDestinations(PRIORITY));
     processDestinations(request, getDestinations(request.getMajor()));
     processDestinations(request, getDestinations(WILDCARD));
}
```

FIG. 26F

```
/**
 * Submit an asynchronous request. See the synchronous
 * requestEventPerformed for more information.
 */
public void requestEventPerformed(RequestEvent request,
RequestResponseListener caller) throws RequestException {
     if (!enabled) {
          throw new RequestException("Transporter disabled");
     }
     if (tagging)
          request.setStatus(request.getStatus() +
               "[Transporter async.]");                                   2608
     //start an inner class thread
     TransporterThread t = new TransporterThread(request, caller);
     runningThreads.put(request, t);
     t.start();
}
```

FIG. 26G

2610

Transporter.TransporterThread:AUS8-1999-0693

```
    /**
    * Private class to handle executions of submits() on another
thread.
    **/
    private class TransporterThread extends Thread {
        /**
        * The current request
        /**
        private RequestEvent request;

        /**
        * The caller of submit that we will call back
        **/
        private RequestResponseListener caller;


        /**
        * Create a transporter thread
        **/
        public TransporterThread(RequestEvent request,
RequestResponseListener caller) {
            super();
            this.request = request;
            this.caller = caller;
        }
        /**
        * Just call the synchronous version of
requestEventPerformed()
        **/
        public void run() {
            try {
                requestEventPerformed(request);
                caller.requestResponse(request);
            } catch (RequestException yikes); {
                caller.requestException(yikes);
            } finally {
                runningThreads.remove(request);
            }
        }
    }
```

FIG. 26H

10700

```
package com.ibm.jtcx.serialization;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
/**
 * Default type comment.
 *
 * <P>INVARIANT:
 */
public class BaseData implements Externalizable {
	private Object[] data = null;
/**
 * BaseData constructor comment.
 */
public BaseData() {
	this(0);
}
/**
 * BaseData constructor comment.
 * @param dataArray java.lang.Object[]
 */
public BaseData(int count) {
	super();

	setData(new Object[count]);
}
/**
 * Default method comment.
 *
 * <P>PRE:
 * <P>POST:
 *
 * @return Parameter not modified
 * @return java.lang.Object[]
 */
public final Object[], getData() {
	return data;
}
```

FIG. 107A

AUS990339US1

10700

```
/**
 * Default method comment.
 *
 * <P>PRE:
 * <P>POST:
 *
 * @return Parameter not modified
 * @return java.lang.Object
 * @param index int
 */
public final Object getData(int index) {
        Object retVal = null;

        if ((data != null) && (index < data.length)) {
                retVal = data[index];
        }

        return retVal;
}
/**
 * Default method comment.
 *
 * <P>PRE:
 * <P>POST:
 *
 * @return Parameter not modified
 * @param in ObjectInput
 */
public void readExternal(ObjectInput in)
        throws ClassNotFoundException, IOException {

        setData((Object[])in.readObject());
}
/**
 * Default method comment.
 *
 * <P>PRE:
 * <P>POST:
 *
 * @return Parameter not modified
 * @param dataArray java.lang.Object[]
 */
public final void setData(Object[] dataArray) {
        data = dataArray;
}
```

FIG. 107B

Aus 9-0339US1

10700

```
/**
 * Default method comment.
 *
 * <P>PRE:
 * <P>POST:
 *
 * @return Parameter not modified
 * @param index int
 * @param dataElement java.lang.Object
 */
public final void setData(int index, object dataElement) {
      if ((data != null) && (index < data.length)) {
            data[index] = dataElement;
      }
}
/**
 * Default method comment.
 *
 * <P>PRE:
 * <P>POST:
 *
 * @return Parameter not modified
 * @param out ObjectOutput
 */
public void writeExternal(ObjectOutput out) throws IOException {
      out.writeObject(getData());
}
```

FIG. 107C

AUS 9[illegible]0339 4511

10800

```
package com.ibm.jtcx.serialization;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.Date;
import java.util.Enumeration;
import java.util.GregorianCalendar;
import java.util.Hashtable;
import java.util.SimpleTimeZone;
import java.util.TimeZone;
import java.util.Vector;
/**
 * Base class of data objects that require small serialization. The
 * attributes of the data object are stored in an array and the elements
 * of the array are written individually.
 *
 * <P>INVARIANT:
 */
public class BaseDataS extends BaseData implements Externalizable {
/**
 * Default constructor.
 */
public BaseDataS() {
        super();
}
/**
 * Creates a new <code>BaseDataS</code> object with a data array of
 * size <code>count</code>.
 *
 * @param count the size of the data array containing the attributes
 */
public BaseDataS(int count) {
        super(count);
```

FIG. 108A

AUS920030339US1

```
}
/**
 * Reads the array of data elements from the stream. The responsibility
 * of reading the individual element is left to the
 * <code>BaseSerializer</code> via <code>readObject()<code>.
 *
 * @param in the input stream that contains the serialized object
 * @exception ClassNotFoundException thrown if
 * <code>BaseSerializer</code> fails to read the object from the stream
 * @exception IOException thrown if
 * <code>BaseSerializer</code> fails to read the object from the stream
 * @see BaseSerializer#readObject
 */
public void readExternal(ObjectInput in)
        throws ClassNotFoundException, IOException {

        int size = in.readShort();

        if (size == -1) {
                setData(null);
        } else {
                Object[] array = new Object[size];

                for (int i = 0; i < size; i++) {
                        array[i] = BaseSerializer.getInstance().readObject(in);
                }

                setData(array);
        }
}
/**
 * Writes the array of data elements. The responsibility of writing the
 * data elements is left to <code>BaseSerializer</code> via
 * <code>writeObject()</code>.
 *
 * @param out the output stream to which the data elements will be
 * written
 */
public void writeExternal(ObjectOutput out) throws IOException {
        Object[] array = getData();

        if (array == null) {
                out.writeShort(-1);
        } else {
                out.writeShort(array.length);

                for (int i = 0; i < array.length; i++) {
                        BaseSerializer.getInstance().writeObject(out, array[i]);
                }
        }
}
```

10800

FIG. 108B

11000

```
package com.ibm.jtcx.serialization;

import java.io.*;
import java.math.BigInteger;
import java.math.BigDecimal;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Hashtable;
import java.util.SimpleTimeZone;
import java.util.StringTokenizer;
import java.util.TimeZone;
import java.util.Vector;

/**
 * The <code>SerializerIF</code> that is used as the base level
 * serializer. It contains three tables used to serialize objects:
 * <br><ul>
 *          <li> codeTable: the table containing the serialization code of
 *                an object based on the name of the class
 *          <li> nameTable: the table containing the name of the class
 *                based on the serialization code
 *          <li> serializationTable: the table containing the serializer of
 *                an object based on its serialization code
 * </ul>
 * <br><br>
 * <code>BaseSerializer</code> delegates the responsibility of
 * serializing the objects to the <code>SerializerIF</code> associated
 * with that class or to the object itself if it implements
 * <code>Externalizable</code>.
 */
public class BaseSerializer implements SerializerIF {
      static private final int NULL_OBJECT = 0;
      static private final int OTHER = 0x00ff;

      static private final String HASHTABLE_SER = "ClassNameHash.ser";
      static private final String INI_FILE = "ClassNames.ini";

      static private Hashtable codeTable = null;
      static private Hashtable nameTable = null;
      static private Hashtable serializerTable = null;
      static private BaseSerializer instance = null;

      class BigDecimalSerializer implements Serializer IF {
            public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
```

FIG. 110A

11000

```
                int scale = in.readShort();
                int size = in.readShort();
                byte[] bytes = new byte[size];
                in.readFully(bytes);

                BigInteger temp = new BigInteger(bytes);
                return new BigDecimal(temp, scale);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                BigDecimal bigD = (BigDecimal)element;

                int scale = bigD.scale();
                bigD.setScale(0);
                byte[] bytes = bigD.toBigInteger().toByteArray();
                bigD.setScale(scale);

                out.writeShort(scale);
                out.writeShort(bytes.length);
                out.write(bytes);
        }
}
class BigIntegerSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException
{
                int size = in.readShort();
                byte[] bytes = new byte[size];
                in.readFully(bytes);

                return new BigInteger(bytes);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                byte[] bytes = ((BigInteger)element).toByteArray();

                out.writeShort(bytes.length);
                out.write(bytes);
        }
}
class BooleanSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException
{
                int value = in.readByte();

                return (value == 1) ? Boolean.TRUE: Boolean.FALSE;
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                out.writeByte(((Boolean)element).booleanValue() ? 1 : 0);
        }
}
```

FIG. 110B

Aus 9[redacted]0339us11

11000

```
class ByteSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
                byte value = in.readByte();

                return new Byte(value);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                out.writeByte(((Byte)element).byteValue());
        }
}
class CharacterSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
                char value = in.readChar();

                return new Character(value);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                out.writeChar(((Character)element).charValue());
        }
}
class DateSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException IOException {
                long value = in.readLong();

                return new Date(value);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                out.writeLong(((Date)element).getTime());
        }
}
class DoubleSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
                double value = in.readDouble();

                return new Double(value);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                out.writeDouble(((Double)element).doubleValue());
        }
}
```

FIG. 110C

AUS 9[illegible]0339US11

11000

```
class FloatSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
                float value = in.readFloat();

                return new Float(value);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                out.writeFloat(((Float)element).floatValue());
        }
}
class GregorianCalendarSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
                long time = in.readLong();
                Date date = new Date(time);
                SerializerIF serializer = BaseSerializer.getInstance();
                TimeZone tz = (TimeZone)serializer.readObject(in);

                GregorianCalender gCalender = new GregorianCalendar(tz);
                gCalendar.setTime(date);

                return gCalendar;
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                GregorianCalendar temp = (GregorianCalendar)element;

                Date date = temp.getTime();
                TimeZone tz = temp.getTimeZone();

                out.writeLong(date.getTime());
                SerializerIF serializer = BaseSerializer.getInstance();
                serializer.writeObject(out, tz);
        }
}
class IntegerSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
                int value = in.readInt();

                return new Integer(value);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
                out.writeInt(((Integer)element).intValue());
        }
}
class LongSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
```

FIG. 110D

11000

```
            long value = in.readLong();

            return new Long(value);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
            out.writeLong(((Long)element).longValue());
        }
}
class ObjectArraySerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
            int size = in.readShort();

            Object[] array = new Object[size];
            for (int i = 0; i < size; i++) {
                SerializerIF serializer = BaseSerializer.getInstance();
                array[i] = serializer.readObject(in);
            }

            return array;
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
            Object[] array = (Object[])element;

            out.writeShort(array.length);
            for (int i = 0; i < array length, i++) {
                SerializerIF serializer = BaseSerializer.getInstance();
                serializer.writeObject(out, array[i];
            }
        }
}
class ObjectSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
            return in.readObject();
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
            out.writeObject(element);
        }
}
class ShortSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
            short value = in.readShort();

            return new Short(value);
        }
```

FIG. 110E

11000

```
        public void writeObject(ObjectOutput out, Object element) throws IOException {
            out.writeShort(((Short)element).shortValue());
        }
}
class SimpleTimeZoneSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
            int offset = in.readInt();
            SerializerIF serializer = BaseSerializer.getInstance();
            String id = (String)serializer.readObject(in);

            return new SimpleTimeZone(offset, id);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
            SimpleTimeZone temp = (SimpleTimeZone)element;

            out.writeInt(temp.getRawOffset());
            SerializerIF serializer = BaseSerializer.getInstance();
            serializer.writeObject(out, temp.getID());
        }
}
class StringSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
            int size = in.readShort();
            byte[] bytes = new byte[size];
            in.readFully(bytes);

            return new String(bytes);
        }
        public void writeObject(ObjectOutput out, Object element) throws IOException {
            byte[] bytes = ((String)element).getBytes();

            out.writeShort(bytes.length);
            out.write(bytes);
        }
}
class VectorSerializer implements SerializerIF {
        public Object readObject(ObjectInput in) throws ClassNotFoundException, IOException {
            int size = in.readShort();

            Vector vector = new Vector(size);
            for (int i = 0; i < size; i++) {
                SerializerIF serializer = BaseSerializer.getInstance();
                vector.addElement(serializer.readObject(in));
            }
```

FIG. 110F

AUS 990339US1

11000

```
                return vector;
            }
            public void writeObject(ObjectOutput out, Object element) throws IOException {
                Vector temp = (Vector)element;

                Object[] array = new Object[temp.size()];
                for (int i = 0; i < array.length; i++) {
                    array[i] = temp.elementAt(i);
                }

                out.writeShort(array.length);
                for (int i = 0; i < array.length; i++) {
                    SerializerIF serializer=BaseSerializer.getInstance();
                    serializer.writeObject(out, array[i]);
                }
            }
        }
    /**
     * Default constructor. The constructor is private because this is a
     * singleton class. When the object is constructed, it initializes its
     * tables.
     */
    private BaseSerializer() {
        init();
    }
    /**
     * Adds the given elements to the three tables.
     *
     * @param className the name of the class
     * @param code the code for the given class
     * @param serializer the object responsible for serializing the given
     * class
     */
    private void addDataToTables(String className, Number code, SerializerIF serializer) {
        getCodeTable().put(code, className);
        getNameTable().put(className, code);

        if (serializer != null) {
            getSerializerTable().put(code, serializer);
        }
    }
```

FIG. 110G

AUS 990339US1

11000

```
/**
 * Creates the codes and serializer objects for the default serialization
 * classes and adds them to the tables. The tables are then written to
 * a serialized file.
 */
private void createDefaultTables() {
      addDataToTables(BigDecimal.class.getName(), new Byte((byte)1), new
BigDecimalSerializer());
      addDataToTables(BigInteger.class.getName(), new Byte((byte)2), new BigIntegerSerializer());
      addDataToTables(Boolean.class.getName(), new Byte((byte)3), new BooleanSerializer());
      addDataToTables(Byte.class.getName(), new Byte((byte)4), new ByteSerializer());
      addDataToTables(Character.class.getName(), new Byte((byte)5), new CharacterSerializer());
      addDataToTables(Date.class.getName(), new Byte((byte)6), new DateSerializer());
      addDataToTables(Double.class.getName(), new Byte((byte)7), new DoubleSerializer());
      addDataToTables(Float.class.getName(), new Byte((byte)8), new FloatSerializer());
      addDataToTables(GregorianCalendar.class.getName(), new Byte((byte)9), new
GregorianCalendarSerializer();
      addDataToTables(Integer.class.getName(), new Byte((byte)10), new IntegerSerializer());
      addDataToTables(Long.class.getName(), new Byte((byte)11), new LongSerializer());
      addDataToTables(Short.class.getName(), new Byte((byte)12), new ShortSerializer());
      addDataToTables(SimpleTimeZone.class.getName(), new Byte((byte)13), new
SimpleTimeZoneSerializer());
      addDataToTables(String.class.getName(), new Byte((byte)14), new StringSerializer());
      addDataToTables(Vector.class.getName(), new Byte((byte)15), new VectorSerializer());
      addDataToTables(Object.class.getName(), new Byte((byte)16), new ObjectSerializer());

      writeTables();
}
/**
 * Returns an instance of the table of class names, keyed by their code.
 * If the table does not exist, it is created.
 *
 * @return The table of class names.
 */
protected Hashtable getCodeTable() {
      if (codeTable == null) {
            codeTable = new Hashtable();
      }
```

FIG. 110H

11000

```
        return codeTable;
    }
    /**
    * Returns an instance of <code>BaseSerializer</code>.
    *
    * @return An instance of <code>BaseSerializer</code>.
    */
    public static SerializerIF getInstance() {
        if (instance == null) {
            instance = new BaseSerializer();
        }

        return instance;
    }
    /**
    * Returns an instance of the table of codes, keyed by their
    * corresponding class name.
    * If the table does not exist, it is created.
    *
    * @return The table of codes.
    */
    protected Hashtable getNameTable() {
        if (nameTable == null) {
            nameTable = new Hashtable();
        }

        return nameTable;
    }
    /**
    * Returns an instance of the table of serializers, keyed by their
    * corresponding code.
    * If the table does not exist, it is created.
    *
    * @ return The table of class names.
    */
    protected Hashtable getSerializerTable() {
        if (serializerTable == null) {
            serializerTable = new Hashtable();
        }

        return serializerTable;
    }
    /**
    * Initializes the hashtable from either a serialized hashtable or from
    * an ini file.
    */
```

FIG. 110I

AUS920030339US1

11000

```
protected void init() {
        File serializedFile = new File(HASHTABLE_SER);
        File iniFile = new File(INI_FILE);

        if (serializedFile.exists()) {
                readSerializedFile(serializedFile);
        } else {
                if (iniFile.exists()) {
                        readIniFile(iniFile);
                }

                createDefaultTables();
        }
}
/**
 * Gets the value of the serialization code from the table based on
 * the className provided. The value returned can either be a
 * <code>Byte</code> or an <code>Integer</code>. The return value
 * will be a <code>Byte</code> if the className is one of the base
 * data types.
 *
 * @return The serialization code of the className.
 * @param className the name of the class
 */
private Number lookupCode(String className) {
        Number code = null;

        if (className != null) {
                code = (Number)getNameTable().get(className);
        }

        return code;
}
/**
 * Looks up the hashcode in the table and returns the String value of
 * the hashcode. If the hashcode does not exist in the table
 * <code>null</code> is returned.
 *
 * @return The object that was stored in the table with the given
 *                hashcode.
 * @param hashcode the hashcode that will be used to look up the value
 */
```

FIG. 110J

11000

```
private String lookupName(Number code) {
        String className = null;

        if (code != null) {
                className = (String)getCodeTable().get(code);
        }

        return className;
}
/**
 * Default method comment.
 *
 * <P>PRE:
 * <P>POST:
 *
 * @return Parameter not modified
 * @return com.ibm.jtc.util.SerializerIF
 * @param code int
 */
private SerializerIF lookupSerializer(Number code) {
        SerializerIF serializer = null;

        if (code != null) {
                serializer = (SerializerIF)getSerializerTable().get(code);
        }

        return serializer;
}
/**
 * Default method comment.
 *
 * <P>PRE:
 * <P>POST:
 *
 * @return Parameter not modified
 * @param iniFile java.io.File
 */
private void readIniFile(File iniFile) {
        BufferedReader in = null;

        try {
                in = new BufferedReader(new FileReader(iniFile));

                for (String inLine = in.readLine(); inLine != null; inLine = in.readLine()) {
                        String trimLine = inLine.trim();
```

FIG. 110K

11000

```
                if ((trimLine.length() > 0) &&
                        !trimLine.startsWith("#")) {
                        StringTokenizer tokenizer = new StringTokenizer(trimLine);

                        String className = tokenizer.nextToken();
                        Integer code = new Integer(className.hashCode());
                        SerializerIF serializer = null;

                        if (tokenizer.hasMoreTokens()) {
                            String serializerName = tokenizer.nextToken();

                            try {
                                serializer = (SerializerIF)Class.forName(serializerName).newInstance();
                            } catch(Exception e) { }
                        }

                        addDataToTables(className, code, serializer);
                }
            }
        } catch (Exception throwAway) {
        } finally {
            try {
                in.close();
            } catch (Exception throwAway) {
            }
        }

        writeTables();
    }
    /**
    * Reads the object from the stream by first reading the code for the
    * element then reads the appropriate data for that object.
    *
    * @return The object that was read from the stream.
    * @param in the input stream that contains the object
    */
    public Object readObject(ObjectInput in)
        throws ClassNotFoundException, IOException {
        Object retVal = null;
        Number code = null;

        byte baseCode = in.readByte();
```

FIG. 110L

AUS990339US1

11000

```
        if (baseCode == NULL_OBJECT) {
                retVal = null;
        } else {
                if (baseCode != OTHER) {
                        code = new Byte(baseCode);
                } else {
                        int secondCode = in.readInt();
                        code = new Integer(secondCode);
                }

                SerializerIF serializer = lookupSerializer(code);
                if (serializer != null) {
                        retVal = serializer.readObject(in);
                } else {
                        String className = lookupName(code);

                        try {
                                retVal = Class.forName(className).newInstance();

                                if (retVal instanceof Externalizable) {
                                        ((Externalizable)retVal).readExternal(in);
                                } else {
                                        retVal = in.readObject();
                                }
                        } catch(Exception e) {
                        }
                }
        }

        return retVal;
}
/**
 * Reads the file containing the serialized hashtables of data.
 *
 * @param serializedFile the file containing the serialized tables
 */
private void readSerializedFile(File serializedFile) {
        ObjectInputStream in = null;
        try {
                in = new ObjectInputStream(new FileInputStream(serializedFile));
                codeTable = (Hashtable)in.readObject();
                nameTable = (Hashtable)in.readObject();
                serializerTable = (Hashtable)in.readObject();
```

FIG. 110M

AUS990339US1

11000

```
        } catch (Exception throwAway) {
        } finally {
                try {
                        in.close();
                } catch (Exception throwAway) { }

                if ((codeTable == null) ||
                        (nameTable == null) ||
                        (serializerTable == null)) {

                        createDefaultTables();
                }
        }
}
/**
 * Writes the given object to the stream. First, the code representing
 * the type of the object is written, then the data within the object
 * is written.
 *
 * @param out the output stream that will contain the object
 * @param element the data object that will be written
 */
public void writeObject(ObjectOutput out, Object element)
        throws IOException {

        if (element == null) {
                out.writeByte(NULL_OBJECT);
        } else {
                String className = element.getClass().getName();
                Number code = lookupCode(className);

                if (code != null) {
                        if (code instanceof Byte) {
                                out.writeByte(code.byteValue());
                        } else if (code instanceof Integer) {
                                out.writeByte(OTHER);
                                out.writeInt(code.intValue());
                        }

                        SerializerIF serializer = lookupSerializer(code);

                        if (serializer != null) {
                                serializer.writeObject(out, element);
                        } else if (element instanceof Externalizable) {
                                ((Externalizable)element).writeExternal(out);
```

FIG. 110N

11000

```
                } else {
                        out.writeObject(element);
                }
        } else {
                if (element instanceof Object[]) {
                        className = Object[].class.getName();
                } else {
                        className = Object.class.getName();
                }

                code = lookupCode(className);
                SerializerIF serializer = lookupSerializer(code);

                out.writeByte(code.byteValue());
                serializer.writeObject(out, element);
        }
    }
}
/**
 * Writes the tables to the file.
 */
private void writeTables() {
    ObjectOutputStream out = null;

    try {
        File serFile = new File(HASHTABLE_SER);
        out = new ObjectOutputStream(new FileOutputStream(serFile));

        out.writeObject(getCodeTable());
        out.writeObject(getNameTable());
        out.writeObject(getSerializerTable());
        out.writeObject(new Date());
    } catch(Exception e) {
    } finally {
        try {
            out.close();
        } catch(Exception e) { }
    }
}
```

FIG. 1100